Lists, Tuples and Dictionaries

Reading Assignment

- Chapter 6 Sections 1, 2, 3 and 4.
- Chapter 8 Section 2.

Chapters Learning Outcomes

At the end of these two chapters, you will be able to

- collect elements using lists
- use the for loop for traversing lists
- learn common algorithms for processing lists
- use lists with functions
- build and use a dictionary container
- work with a dictionary for table lookups

Basic Properties of Lists

Motivation

- Assume that you are given 10 values to store in a program for later processing (e.g. finding the largest element).
- One way to achieve this is to use 10 variables `value1`, `value2`, ..., `value10`.

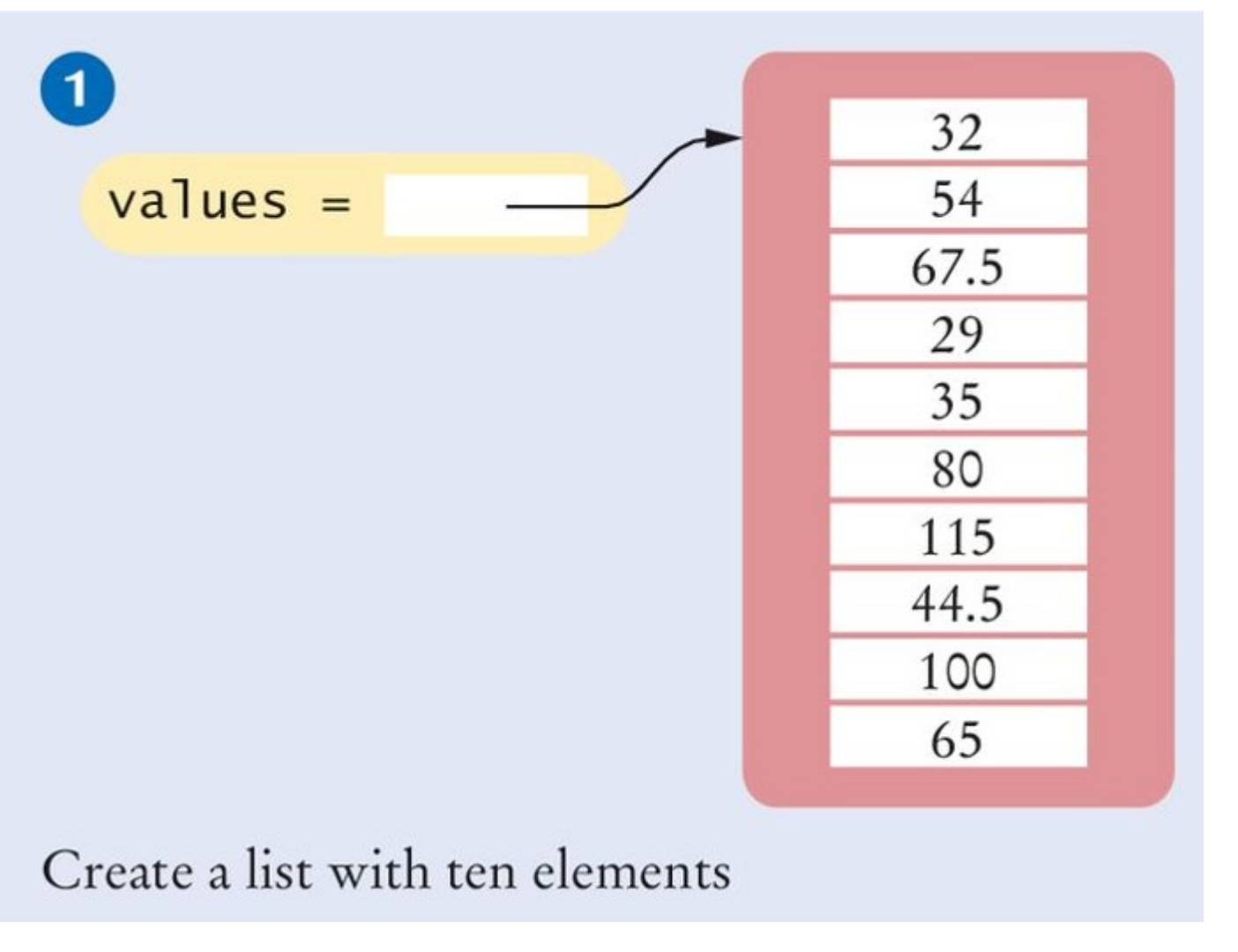
- 32
- 54
- 67.5
- 29
- 35
- 80
- 115
- 44.5
- 100
- 65

- However, such a sequence of variables is not very practical to use. Why?
  - Because ...
- That is why we can use lists to overcome this difficulty.

- However, such a sequence of variables is not very practical to use. Why?
  - Because ***You will need to repeat the same code 10 times.***
  - Why can't we include it in a for loop?
  - We can't change the variable names inside the loop.

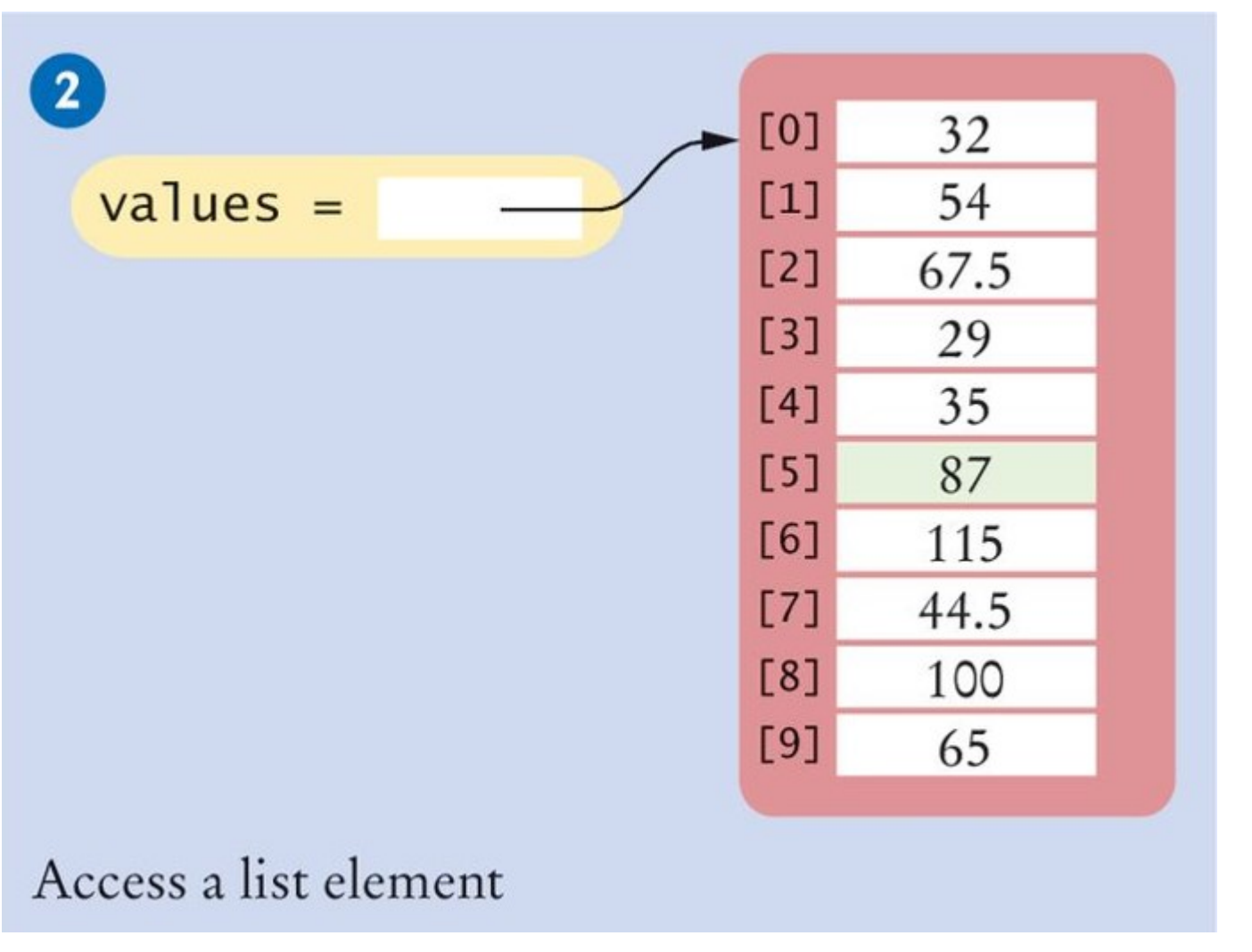
Creating Lists

- `values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]`
  - The square brackets indicate that we are creating a list.
  - The items are stored in the order they are provided.



Accessing List Elements

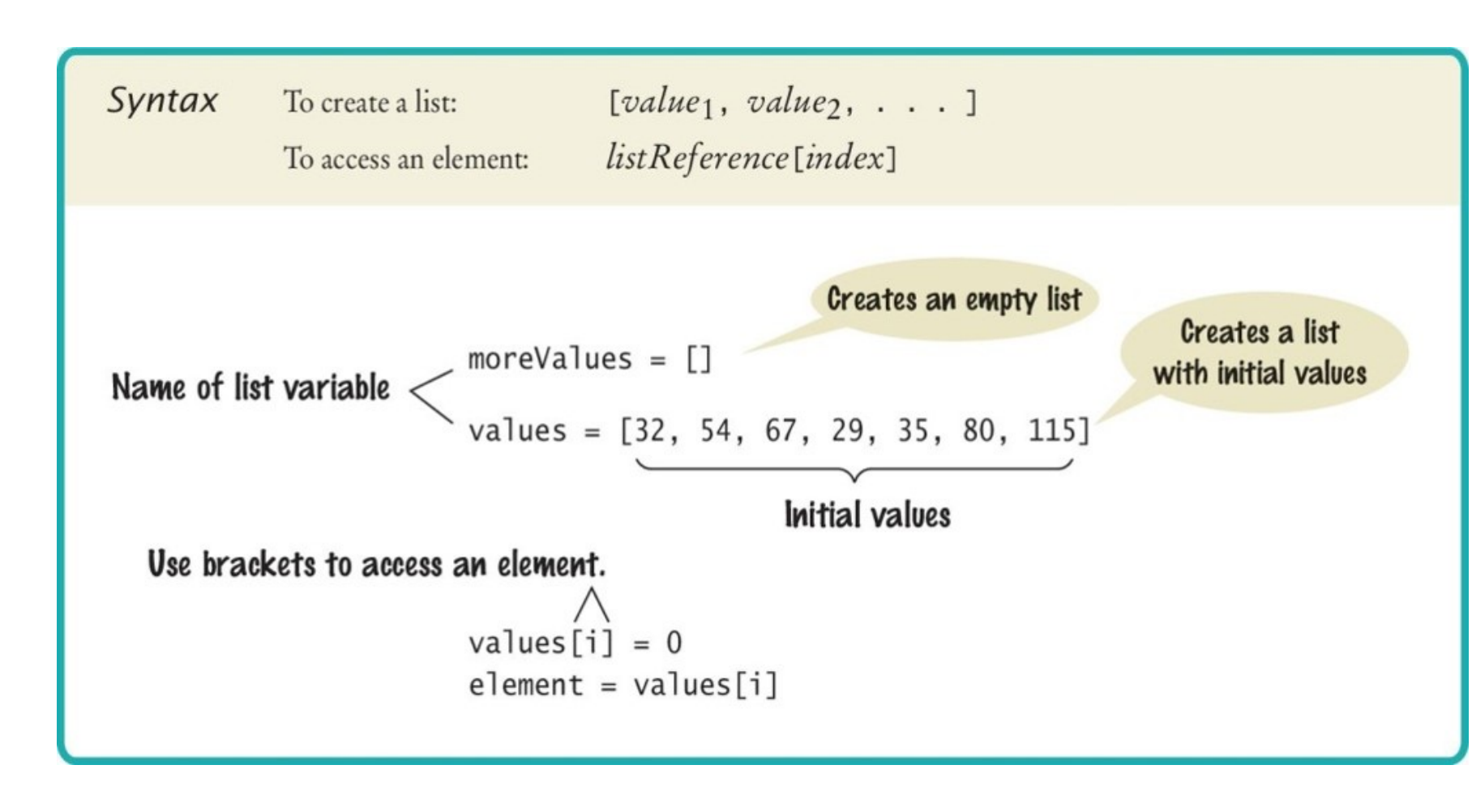
- A list is a sequence of elements, each of which has an integer position or index
- To access a list element, you specify which index you want to use.
  - This is done with the subscript operator in the same way that you access individual characters in a **string**.



In [ ]:
```
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]
values[5]=87
print(values[5])
print(type(values[2]))
print(type(values[3]))
```

Differences between Lists and Strings

- There are two differences between **lists** and **strings**
  - Lists can hold values of any type, whereas strings are sequences of characters.
  - Strings are immutable — you cannot change the characters in the sequence. But lists are mutable.



- What is the difference between the following?
  - `values[4]`
  - `values = [4]`

- When accessing a variable in a list, the index of the list must stay within the valid range.
  - Otherwise, an **out-of-range** error will result from using an index not in the range.

In [ ]:
```
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]
values[10]=87
```

- One can use the **`len`** function to obtain the length of the list.

List Traversal

- **List traversal** refers to visiting (and may be processing) each element in the list once.

- There are two ways to traverse a list:

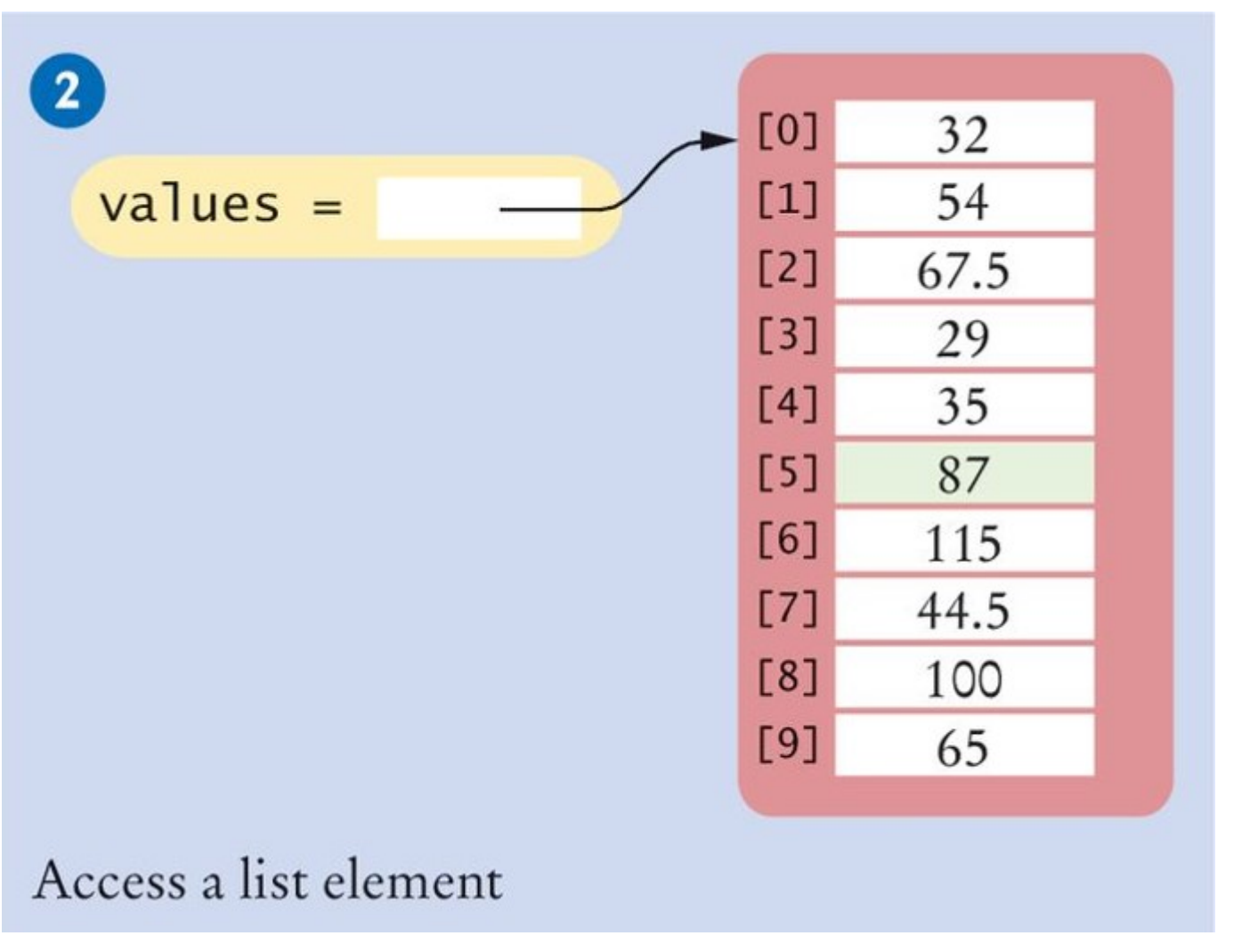
In [ ]:
```
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]
```

In [ ]:
```
# You have access to index values and elements
for i in range(len(values)) :
 print(i, values[i])
```

In [ ]:
```
# You only have access to elements
for element in values :
 print(element)
```

List References

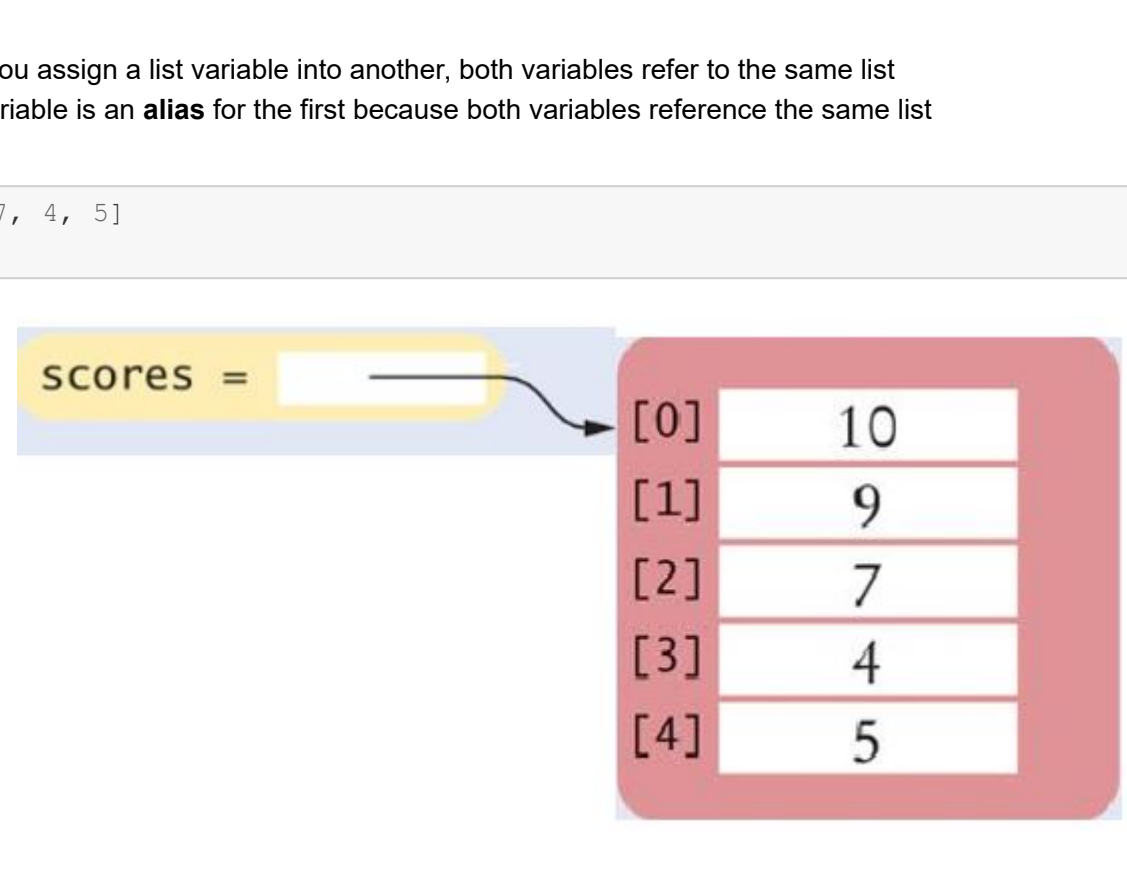
- Make sure you see the difference between the:
  - List variable: The named *alias* or pointer to the list
  - List contents: Memory where the values are stored
    - which is usually elsewhere

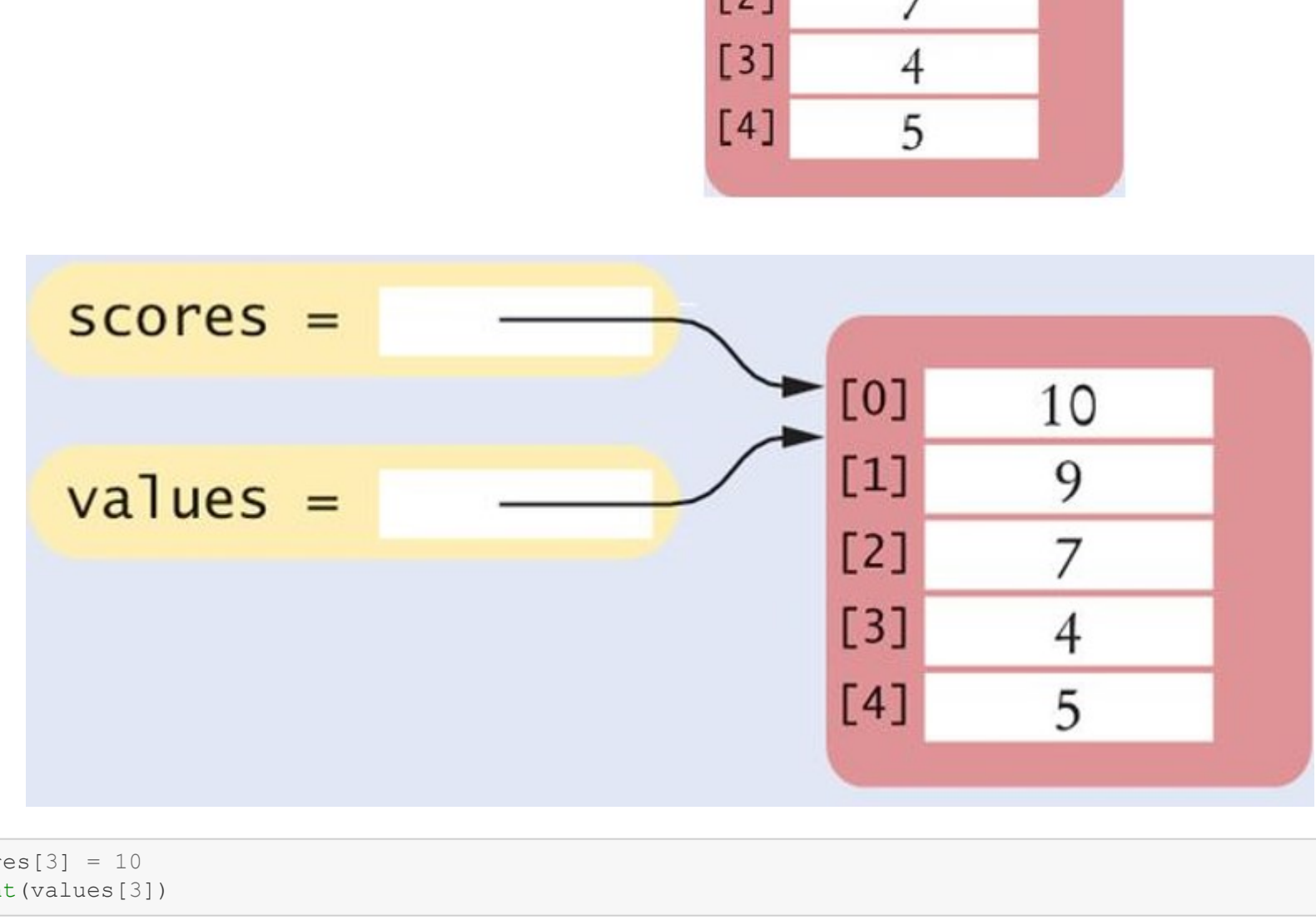


- A list variable contains a **reference** to the list contents.
- The **reference** is the location of the list contents (in memory).

- That is why when you assign a list variable into another, both variables refer to the same list
  - The second variable is an **alias** for the first because both variables reference the same list

In [ ]:
```
scores = [10, 9, 7, 4, 5]
values = scores
```





In [ ]:
```
scores[3] = 10
print(values[3])
```

Student Activity

- Define a list of integers, `primes`, containing the first five prime numbers.
- What does the list `primes` contain after executing the following loop.

In [ ]:
```
## Student Activity
# primes
for i in range(2) :
 primes[4 - i] = primes[i]
```

List Operations

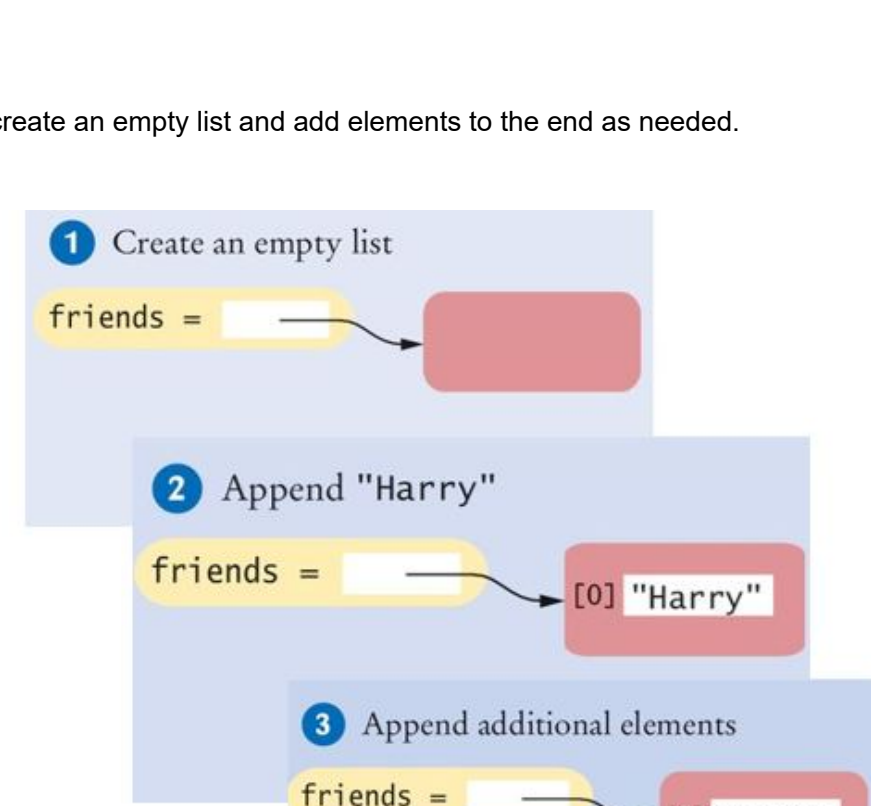
Appending Elements

- If we do not know all the elements of a list, beforehand, we can create an empty list and add elements to the end as needed.

```
#1
friends = []

#2
friends.append("Harry")


#3
friends.append("Emily")
friends.append("Bob")
friends.append("Cari")
```

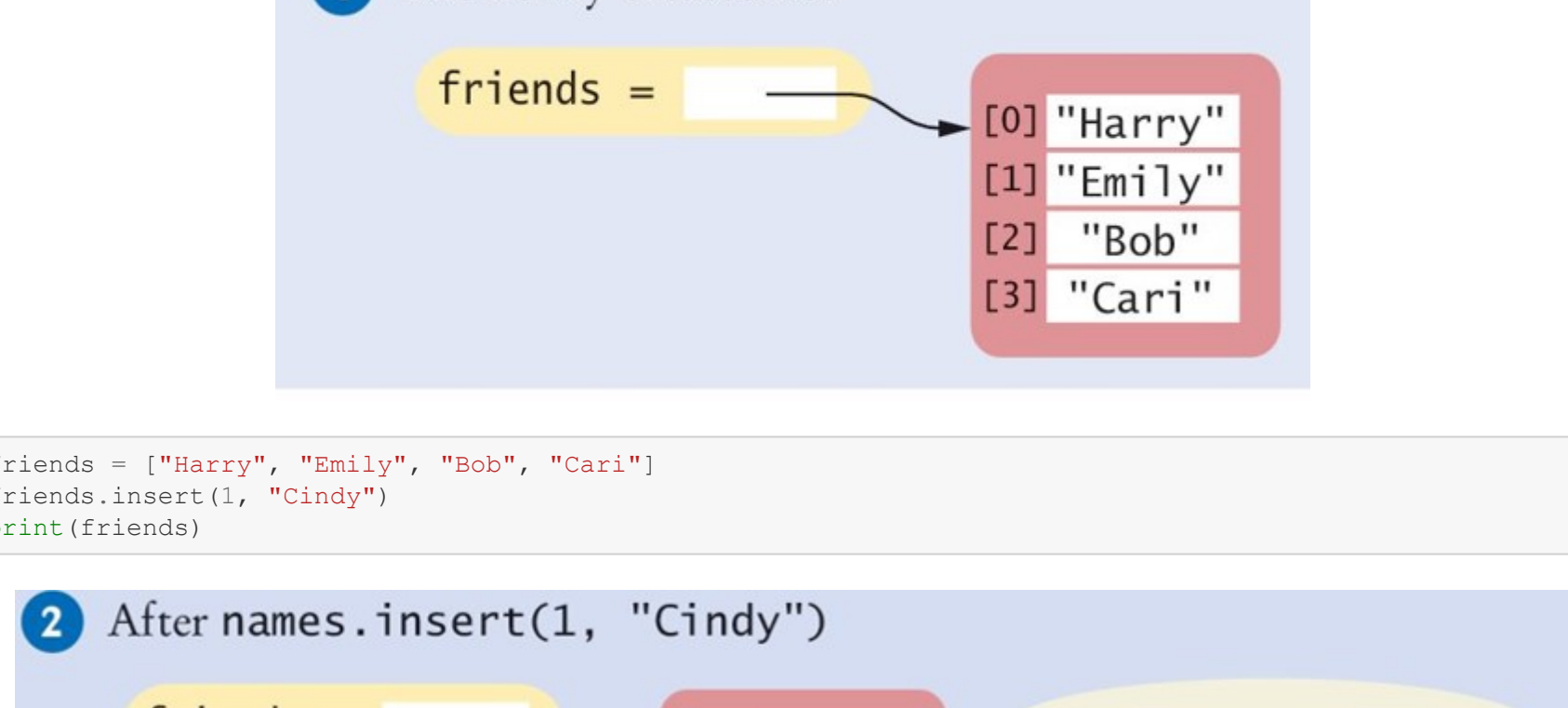


Inserting Elements

- Sometimes the order in which elements are added to a list is important
- A new element has to be inserted at a specific position in the list
- e.g.,

In [ ]:
```
friends = ["Harry", "Emily", "Bob", "Cari"]
```

In [ ]:
```
friends = ["Harry", "Emily", "Bob", "Cari"]
friends.insert(1, "Cindy")
print(friends)
```



- Note that the index at which the new element is to be inserted must be between 0 and the number of elements currently in the list.

Finding an Element

- We can determine whether the element is in the list or not

In [ ]:
```
friends = ["Harry", "Emily", "Bob", "Cari", "Emily"]
if "Cindy" in friends :
    print("She's a friend")
```

- We can, further, determine the index of the element if it is in the list.

In [ ]:
```
friends = ["Harry", "Emily", "Bob", "Cari", "Emily"]
n = friends.index("Emily")
print(n)
```

Removing an Element

- The `pop()` method removes the element at a given position.

In [ ]:
```
friends = ["Harry", "Cindy", "Emily", "Bob", "Cari","Bill"]
friends.pop(1)
print(friends)
```

- The `pop()` method without an argument will remove the last element in the list.

In [ ]:
```
friends = ["Harry", "Cindy", "Emily", "Bob", "Cari","Bill"]
friends.pop()
print(friends)
```

- The element removed from the list is **returned** by the pop() method.

In [ ]:
```
friends = ["Harry", "Cindy", "Emily", "Bob", "Cari","Bill"]
print(friends.pop())
print(friends)
```

- The remove method removes an element by ***value*** instead of by ***position***.

In [ ]:
```
friends = ["Harry", "Cindy", "Emily", "Bob", "Cari","Bill"]
friends.remove("Cari")
print(friends)
```

- Note that the value being removed must be in the list or an exception is raised.

Concatenation and Replication

- The concatenation of two lists is a new list that contains the elements of the first list, followed by the elements of the second.
- This is accomplished using the concatenation operator `+`.

In [ ]:
```
myFriends = ["Fritz", "Cindy"]
yourFriends = ["Lee", "Pat", "Phuong"]
ourFriends = myFriends + yourFriends
print(ourFriends)
```

- If you want to concatenate the same list multiple times, use the replication operator `*`.

In [ ]:
```
# Repeating the same list multiple times
monthInQurater = [1, 2, 3] * 4
print(monthInQurater)
```

In [ ]:
```
# Replication can be used to initialize a list with a fixed value
monthlyScores = [0] * 12
print(monthlyScores)
```

Equality Testing

- You can use the == operator to compare whether two lists have the same elements, in the same order.
- Similarly, you can use the != operator to compare whether two lists are different.

In [ ]:
```
l1 = [1, 4, 9] == [1, 4, 9]      # Result is
l2 = [1, 4, 9 ] == [4, 1, 9]     # Result is
l3 = [1, 4, 9] != [1, 4]         # Result is
```

In [ ]:
```
print("l1 is", l1, ", l2 is", l2, "and l3 is", l3)
```

Copying Lists

- Given a list **`values`**, if we want to make a copy of it, does the following work?

In [ ]:
```
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]
anotherCopy = values
print(values)
print(anotherCopy)
anotherCopy[2] = -1
print(values)
print(anotherCopy)
```

- In order to make a copy of the list, use the **`list()`** function.

In [ ]:
```
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]
anotherCopy = list(values)
print(values)
print(anotherCopy)
anotherCopy[2] = -1
print(values)
print(anotherCopy)
```

Common List Algorithms

- In this section, we will see different tasks that cannot, in general, be performed using a pre-defined library function.
- Refer to the textbook for more examples.

Filling

- Write Python code that will generate a list of squares (0, 1, 4, 9, 16, ..., 100).

In [ ]:
```
values = []
for i in range(11) :
 values.append(i * i)
print(values)
```

Linear Search

- Write Python code to find the index of the first value greater than 100 in a list of numbers. Your code should display `Not Found` if such element does not exist.

In [ ]:
```
values = [-1, 5, 19, 22, 33, 106] * 7
limit = 100
pos = 0
found = False
while pos < len(values) and not found :
    if values[pos] > limit :
        found = True
    else :
        pos = pos + 1

if found :
    print("Found at position:", pos)
else :
    print("Not found")
```

Swapping Elements

- Given a list of values, `values`, swap the first element with the last element
- Does the following code work?

In [ ]:
```
values = [-1, 5, 19, 22, 33, 106]
values[0] = values[len(values)-1]
values[len(values)-1] = values[0]
```

In [ ]:
```
print(values)
```

- What we need to do is store the first value **somewhere** before assigning it to the other element.

In [ ]:
```
values = [-1, 5, 19, 22, 33, 106]
temp = values[0]
values[0] = values[len(values)-1]
values[len(values)-1] = temp
print(values)
```

Using Lists with Functions

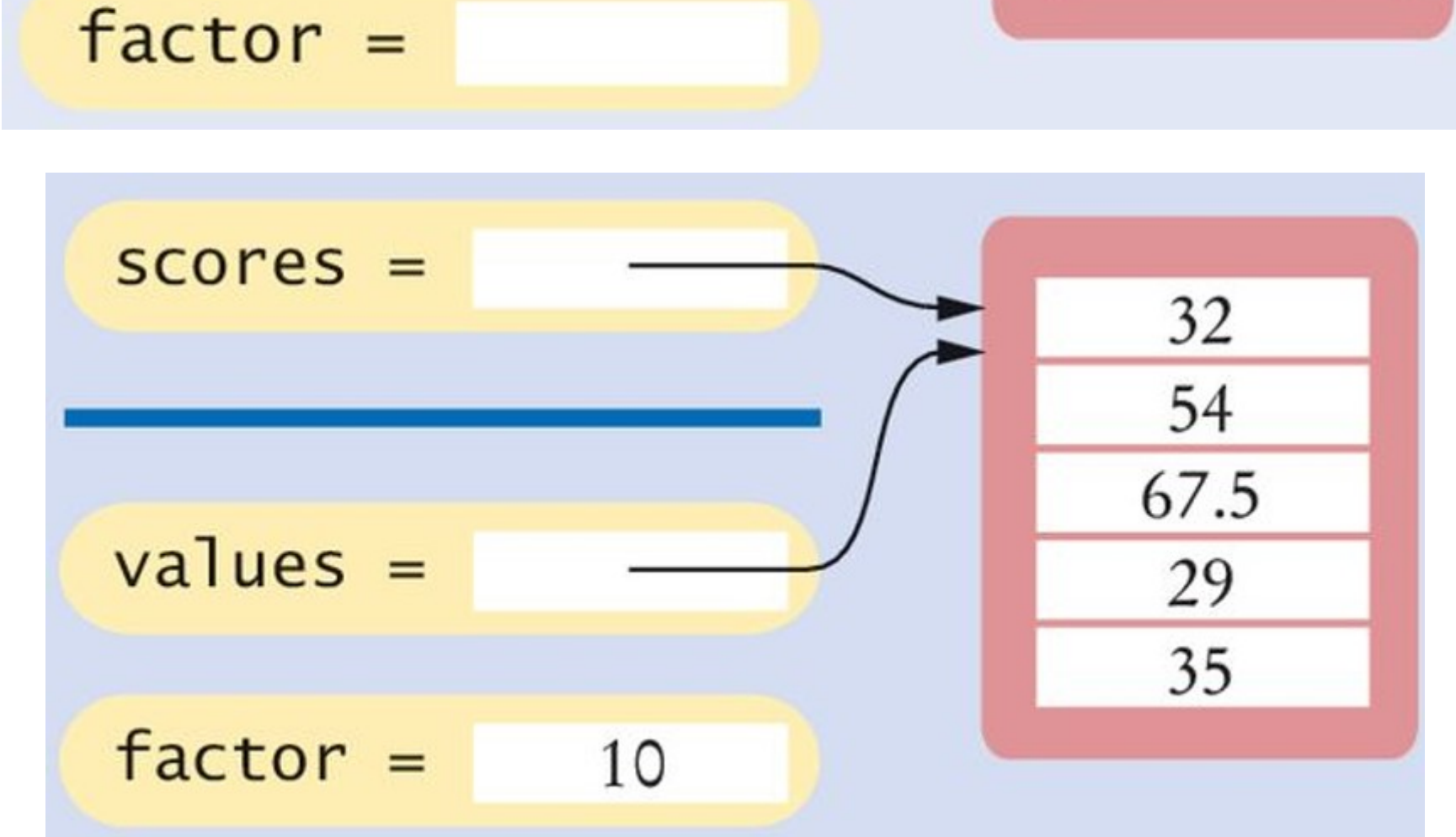
- A function can accept a list as an argument.
- The following function multiplies all elements of a list by a given factor:

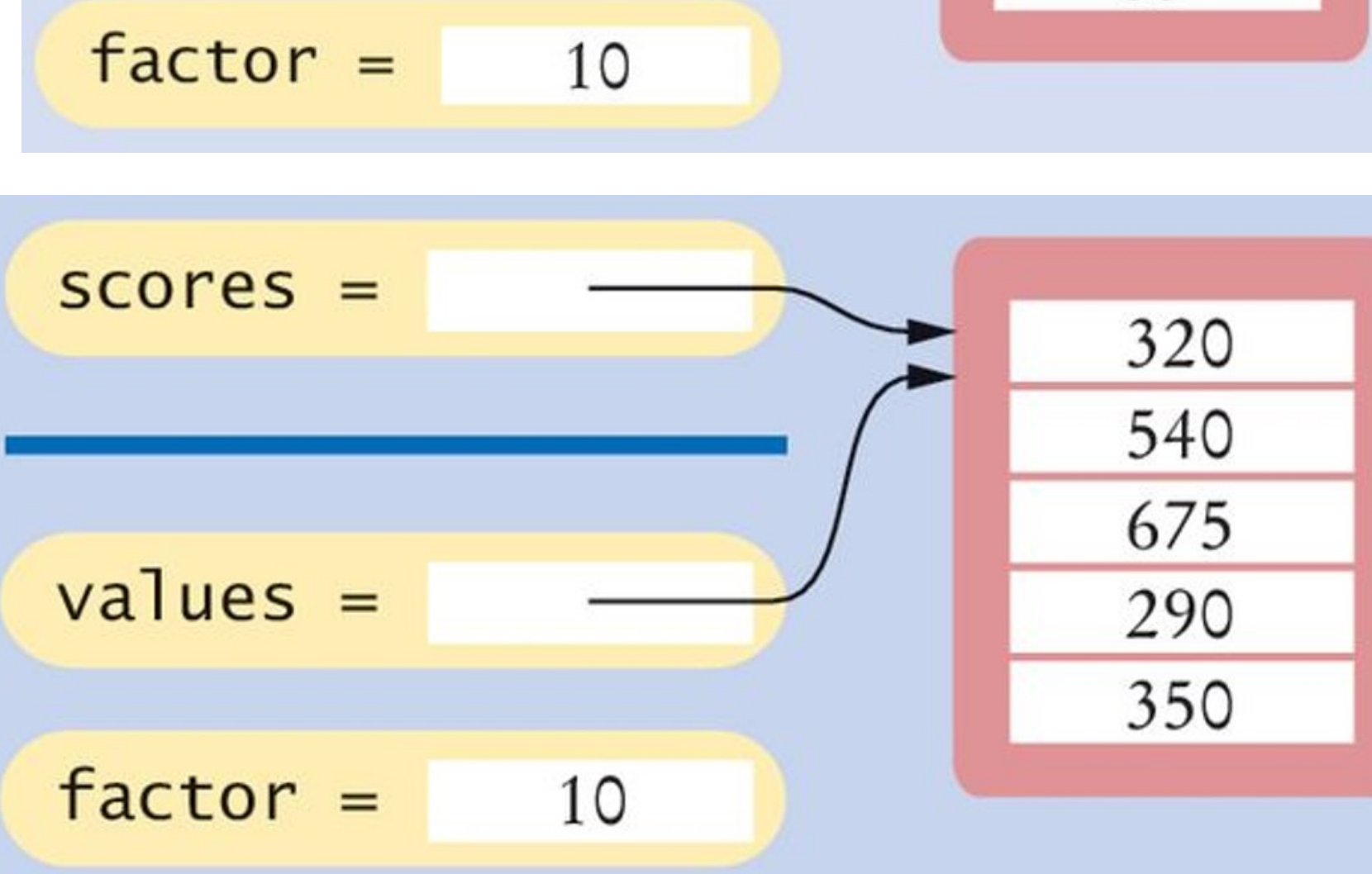
In [ ]:
```
def multiply(values, factor) :
    for i in range(len(values)) :
        values[i] = values[i] * factor
scores = [32, 54, 67.5, 29, 35]
multiply (scores, 10)
```

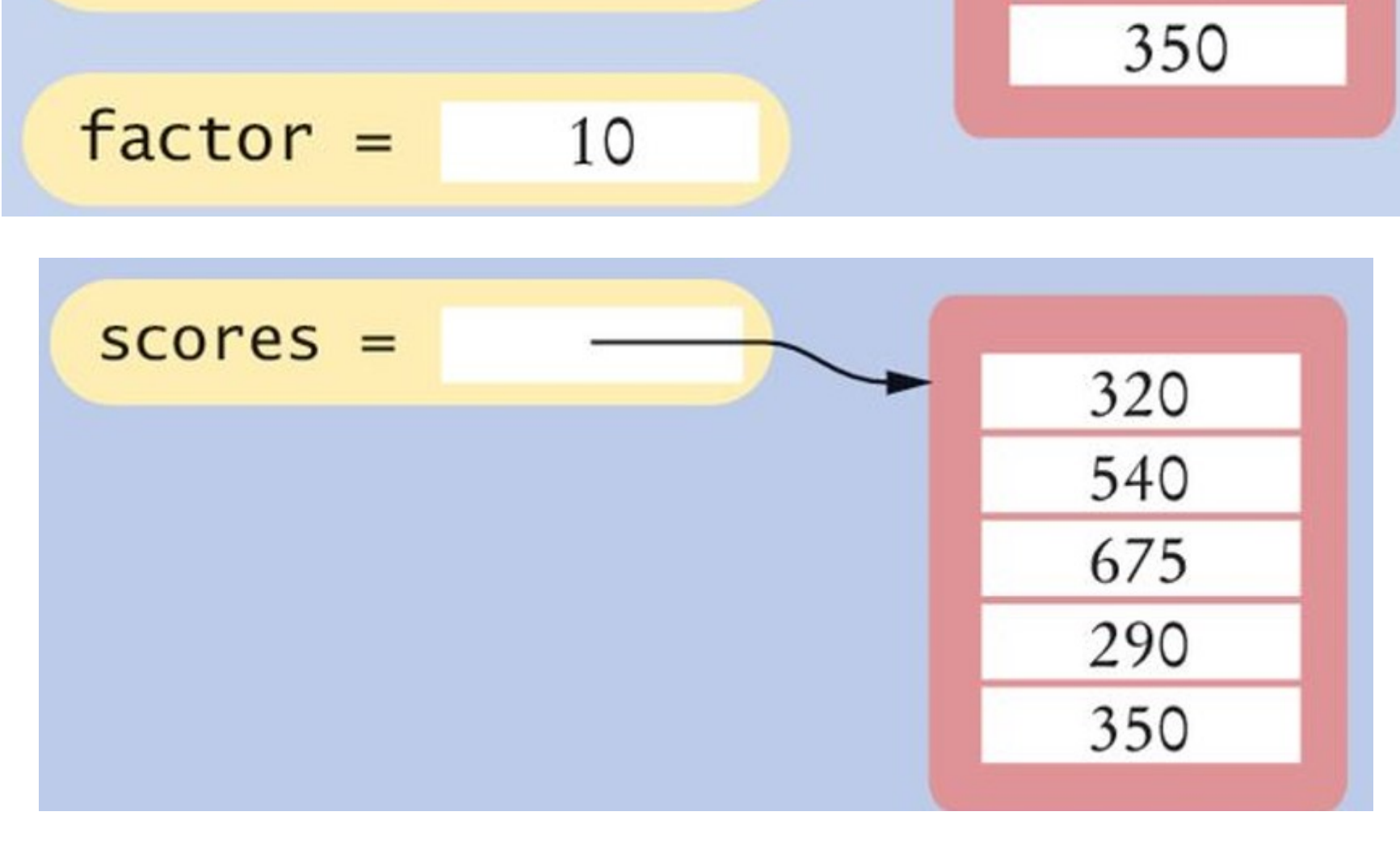
In [ ]:
```
# Do you think that the values of the list scores will change?
print(scores)
```

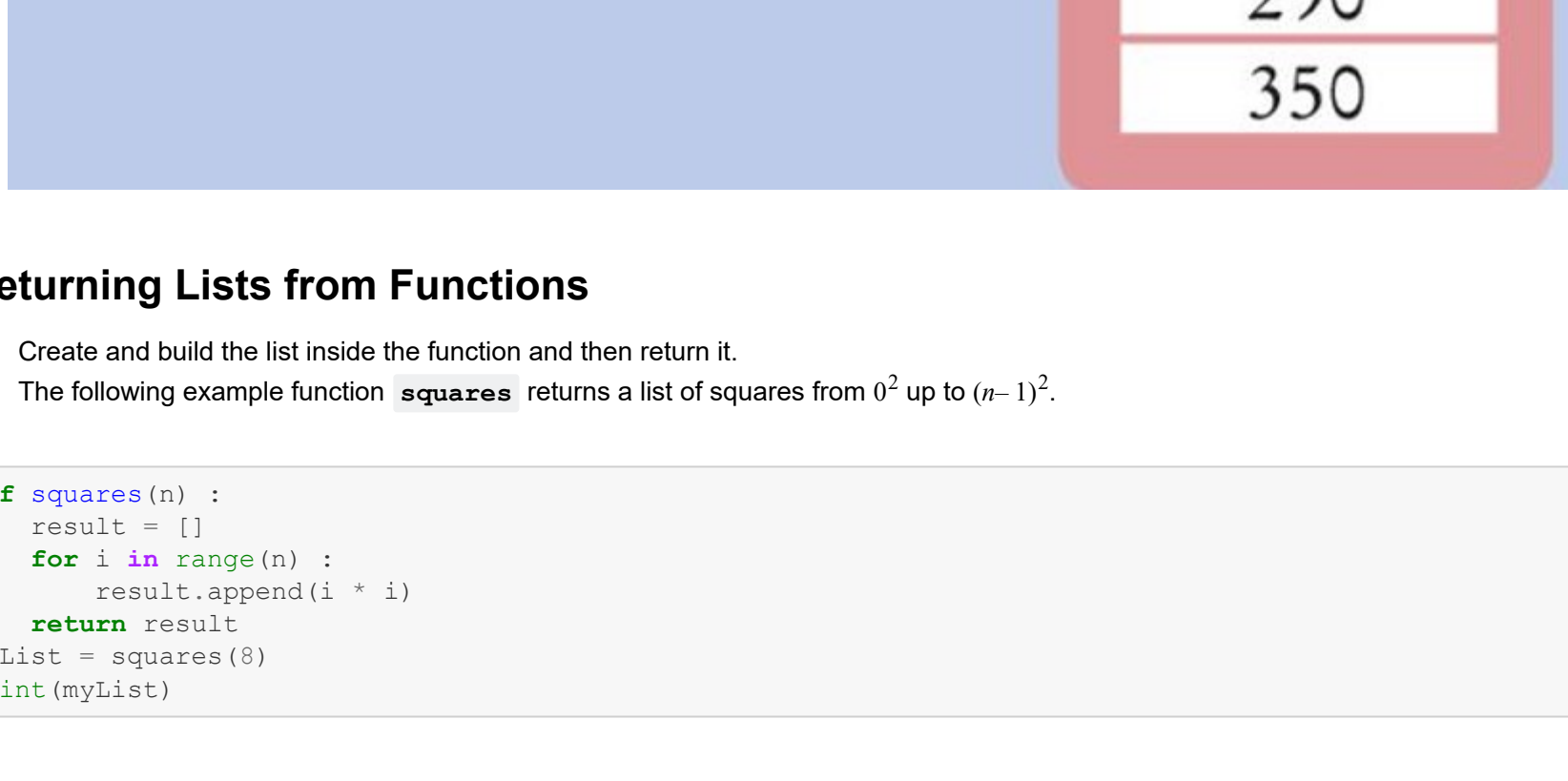
- The answer is yes. The reason is that `values` has a copy of the reference to the `scores` list.

- Let us see exactly what happened.









Returning Lists from Functions

- Create and build the list inside the function and then return it.
- The following example function **`squares`** returns a list of squares from 0^2 up to $(n-1)^2$.

In [ ]:
```
def squares(n) :
    result = []
    for i in range(n) :
        result.append(i * i)
    return result
myList = squares(8)
print(myList)
```

Tuples

- A tuple is similar to a list, but once created, its contents cannot be modified.
  - i.e., a tuple is an immutable version of a list.

- A tuple is created by specifying its contents as a comma-separated sequence.
- You can either enclose the sequence in parentheses or omit them.

In [ ]:
```
triple1 = (5, 10, 15)
triple2 = 5, 10, 15
print(triple1==triple2)
```

- Tuples are commonly used to return **multiple values** from functions

In [ ]:
```
# Function definition
def readDate() :
    print("Enter a date:")
    month = int(input(" month: "))
    day = int(input(" day: "))
    year = int(input(" year: "))
    return (month, day, year) # Returns a tuple.

# Function call: assign entire value to a tuple
date = readDate()
print(date)

# Function call: use tuple assignment:
(month, day, year)  = readDate()
print(day, "/", month,"/", year)
```

Dictionaries

- A dictionary is a container that keeps associations between **keys** and **values**.
  - Every key in the dictionary has an associated value.

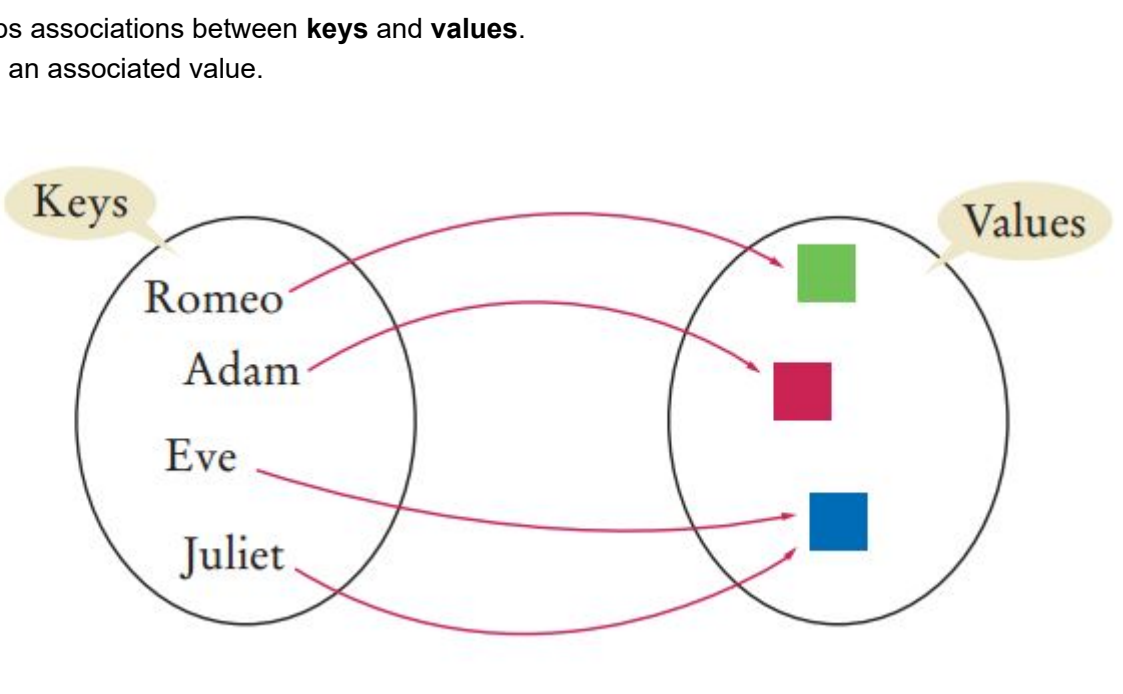


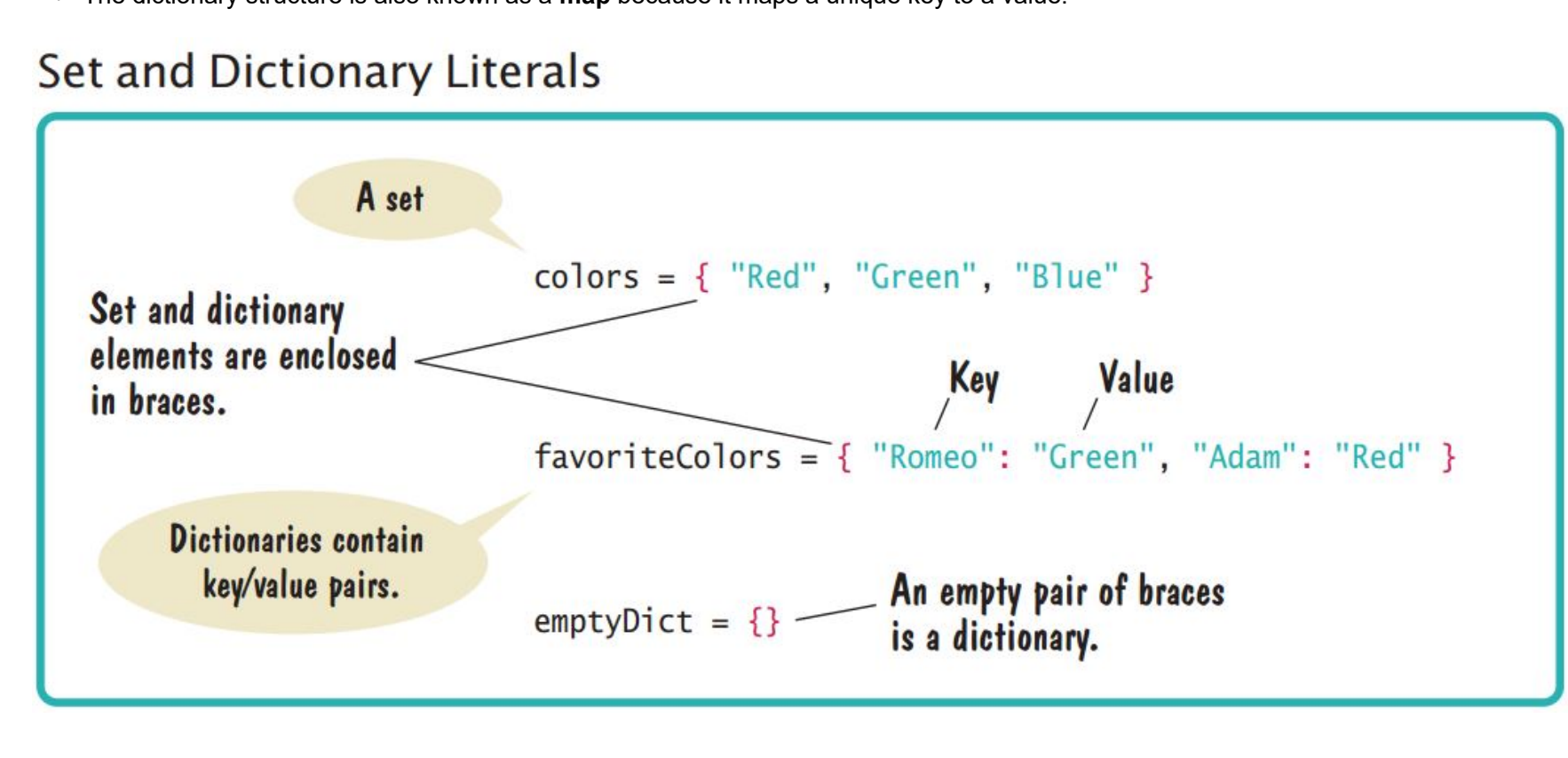
Figure 7
A Dictionary

- Keys are unique, but a value may be associated with several keys.

In [ ]:
```
favoriteColors = { "Romeo": "Green", "Adam": "Red", "John": "Blue", "Sam": "Red" }
print(favoriteColors)
```

- The dictionary structure is also known as a **map** because it maps a unique key to a value.

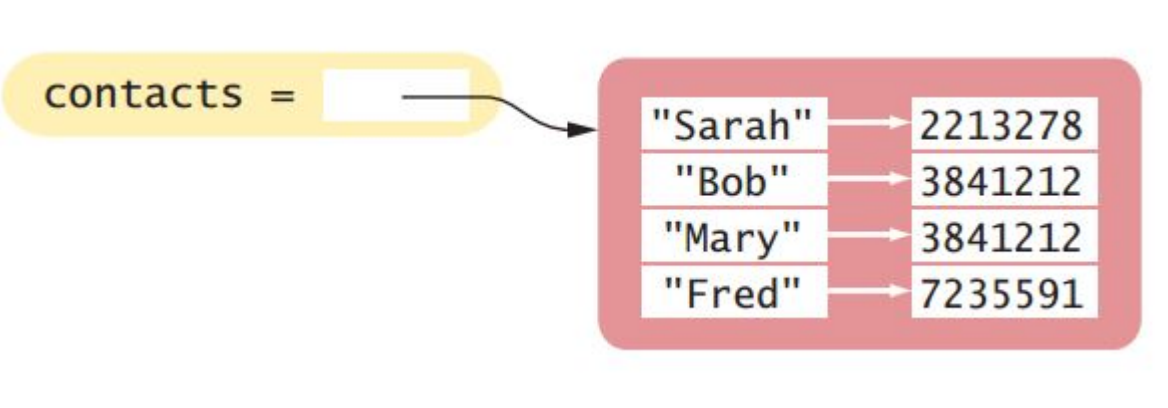
Set and Dictionary Literals



Creating Dictionaries

- A dictionary is created where each key:value pair is separated by a colon.
- The collection of key:value pairs are enclosed in braces.
- Note that when the braces contain key:value pairs, they denote a dictionary.
  - Otherwise, if it contains elements separated by commas, it is considered a set.
- The only ambiguous case is an empty {}. By convention, it denotes an empty dictionary, not an empty set.

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob": 3841212, "Sarah": 2213278 }
print(contacts)
```



- You can create a duplicate copy of a dictionary using the `dict` function:

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob": 3841212, "Sarah": 2213278 }
oldContacts = dict(contacts)
print(oldContacts)
```

Accessing Dictionary Values

- The subscript operator [] is used to return the value associated with a key.

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob": 3841212, "Sarah": 2213278 }
print("Bob's number is", contacts["Bob"])
```

- Are the **key:value** pairs of the dictionary ordered?

- The answer is No. You cannot access the items by index or position.
  - A **value** can only be accessed using its associated **key**.

- Note that the key supplied to the subscript operator must be a valid key in the dictionary, otherwise a `KeyError` exception will be raised.

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob": 3841212, "Sarah": 2213278 }
myContact = input("Enter the contact name: ")
if myContact in contacts :
    print(myContact + "'s number is", contacts[myContact])
else :
    print(myContact + " is not in my contact list.")
```

- Instead of using an `if` statement to make sure that a contact exists before you get its associated value, you can use the `get` method and supply it with a *default value* to return, if the key was not in the dictionary.

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob": 3841212, "Sarah": 2213278 }
myContact = input("Enter the contact name: ")
myNumber = contacts.get(myContact, 411)
print("The phone number of " + myContact + " is " + str(myNumber))
```

Adding and Modifying Items in a Dictionary

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob": 3841212, "Sarah": 2213278 }
contacts["John"] = 2034847 # Adding a new contact
print(contacts)
contacts["John"] = 2037663 # Modifying the value of a contact
print(contacts)
```

In [ ]:
```
favoriteColors = {}                  # In case we do not know the elements of the dictionary beforehand
favoriteColors["Juliet"] = "Blue"    # Add a key:value pair to the dictionary
favoriteColors["Adam"] = "Red"       # Add a key:value pair to the dictionary
favoriteColors["Eve"] = "Blue"       # Add a key:value pair to the dictionary
favoriteColors["Romeo"] = "Green"    # Add a key:value pair to the dictionary
print(favoriteColors)
```

Removing a key from a Dictionary

- Call the `pop` method with the key as the argument.
- `pop` returns the value of the removed key.
- A `KeyError` exception is raised if the key to be removed is not in the dictionary.

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob": 3841212, "Sarah": 2213278 }
myContact = input("Enter the contact name to remove: ")
contactsNumber = contacts.pop(myContact)
print(myContact + " was removed, and the current contacts contain the following\n", contacts)
```

Traversing a Dictionary

- To access the value associated with a key in the body of the loop, you can use the loop variable with the subscript operator.

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob": 3841212, "Sarah": 2213278 }
print("My Contacts:")
for key in contacts :
    print("%-10s %d" % (key, contacts[key] ))
```

- To iterate through the keys in sorted order, you can use the `sorted` function as part of the for loop:

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob": 3841212, "Sarah": 2213278 }
print("My Contacts:")
for key in sorted(contacts) :
    print("%-10s %d" % (key, contacts[key] ))
```

- You can also iterate over the values of the items, instead of the keys, using the `values` method.
- This can be useful for creating a list that contains all of the phone numbers in our dictionary:

In [ ]:
```
contacts = { "Fred": 7235591, "Mary": 3841212,
            "Bob": 3841212, "Sarah": 2213278 }
phoneNumbers = [] # Create an empty list.
for number in contacts.values()  :
    phoneNumbers.append(number)
print(phoneNumbers)
```

- This can also be achieved through the `list` function.

In [ ]:
```
samePhoneNumbers = list(contacts.values())
print(samePhoneNumbers)
```

Summary

- Use lists for collecting values.
- Know and use the built-in operations for lists.
- Know and use common list algorithms.
- Implement functions that process lists.
- Work with Python dictionaries.